CS2100-lecture-sorting: A tour of the top 5 sorting algorithms with Python code

#utec/CS2100 #utec/CS2100/lecture

Algorithm

One of the seemingly most-overused words in tech is "algorithm". From the apps on your phone to the sensors in your wearables and how posts appear in your Facebook News Feed, you'll be pushed to find a service that isn't powered by some form of algorithm. I am too fascinated how algorithms made an impact in our day-to-day lives

> An algorithm is a finite sequence of precise instructions for performing a computation or for solving a problem.

For example, here is an algorithm for singing that annoying song "99 Bottles of Beer on the Wall", for arbitrary values of 99:

```
BottlesOfBeer(n):
  For i ← n down to 1
      Sing "i bottles of beer on the wall, i bottles of beer,"
      Sing "Take one down, pass it around, i − 1 bottles of beer on the wall."
  Sing "No bottles of beer on the wall, no bottles of beer,"
  Sing "Go to the store, buy some more, n bottles of beer on the wall."
```

Properties of Algorithms

- **Input** -An algorithm has input values from a specified set.
- **Output** -From each set of input values an algorithm produces output values from a specified set. The output values are the solution to the problem.
- **Definiteness** -The steps of an algorithm must be defined precisely.
- **Correctness**-An algorithm should produce the correct output values for each set of input values.
- **Finiteness**-An algorithm should produce the desired output after a finite (but perhaps large) number of steps for any input in the set.
- **Effectiveness**-It must be possible to perform each step of an algorithm exactly and in a finite amount of time.
- **Generality**-The procedure should be applicable for all problems of the desired form, not just for a particular set of input values

Now before heading up to main topic, I want to share the basics of analysis of the

algorithms including time complexity and space complexity.

Time Complexity and Space Complexity in algorithms

Always a question arises -

> When does an algorithm provide a satisfactory solution to a problem?

- One measure of efficiency is the time used by a computer to solve a problem using the algorithm, when input values are of a specified size
- Second measure is the amount of computer memory required to implement the algorithm when input values are of a specified size

Questions such as these involve the **computational complexity** of the algorithm. An analysis of the time required to solve a problem of a particular size involves the **time complexity** of the algorithm. An analysis of the computer memory required involves the **space complexity** of the algorithm.

There are three types of time complexity –**Best, average and worst case**.
In simple words for an algorithm, if we could perform and get what we want in just one(eg. on first instance) **computational**approach, then that is said as **O(1) i.e. Time complexity here falls into "Best case" category.**

Say for example, same algorithm results into many **iterations/recursions** or say n times it had to perform to get the result, then the example used for this algorithm describes it's **worst case time complexity.**

Below are some common time complexities with simple definitions.

- ***Constant time*** has an order of growth `1`, for example: `a = b + c`.
- ***Logarithmic time*** has an order of growth `LogN`, it usually occurs when you're dividing something in half (binary search, trees, even loops), or multiplying something in same way.
- ***Linear***, order of growth is `N`, for example

```
int p = 0;
for (int i = 1; i < N; i++)
  p = p + 2;
```

- ***Linearithmic***, order of growth is `n*logN`, usually occurs in divide and conquer algorithms.
- ***Cubic***, order of growth `N^3`, classic example is a triple loop where you check all triplets:

```
int x = 0;
for (int i = 0; i < N; i++)
   for (int j = 0; j < N; j++)
      for (int k = 0; k < N; k++)
          x = x + 2
```

- ***Exponential***, order of growth `2^N`, usually occurs when you do exhaustive search, for example check subsets of some set.

Simple example with code -

So scenario on time complexity for this above given example would be -

int i=0;	This will be executed only once. The time is actually calculated to i=0 and not the declaration.
i<N;	This will be executed N+1 times
i++ ;	This will be executed N times
if(arr[i]!=invalidChar)	This will be executed N times
arr[ptr]=arr[i];	This will be executed N times (in worst case scenario)
ptr++;	This will be executed N times (in worst case scenario)

Asymptotic Notations

Asymptotic Notations are languages that allow us to analyze an algorithm's running time by identifying its behavior as the input size for the algorithm increases. This is also known as an algorithm's growth rate.
The following 3 asymptotic notations are mostly used to represent time complexity of algorithms:

Big Oh (O)

Big Oh is often used to describe the worst-case of an algorithm by taking the highest order of a polynomial function and ignoring all the constants value since they aren't too influential for sufficiently large input.

Big Omega (Ω)

Big Omega is the opposite of Big Oh, if Big Oh was used to describe the upper bound (worst-case) of a asymptotic function, Big Omega is used to describe the lower bound of a asymptotic function. In analysis algorithm, this notation is usually used to describe the

complexity of an algorithm in the best-case, which means the algorithm will not be better than its **best-case**.

Big Theta (Θ)

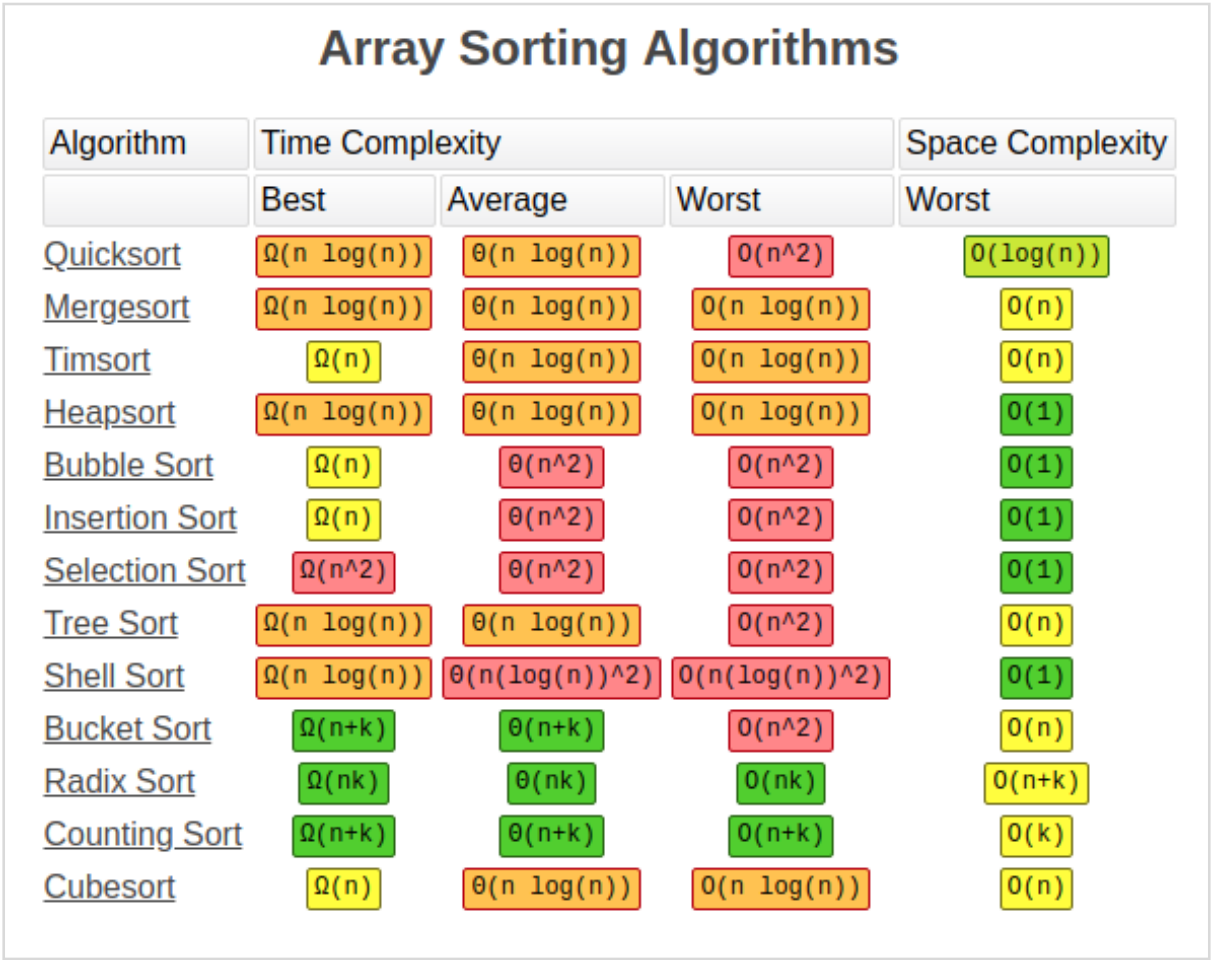
When an algorithm has a complexity with lower bound = upper bound, say that an algorithm has a complexity O(n log n) and Ω(n log n), it's actually has the complexity Θ(n log n), which means the running time of that algorithm always falls in n log n in the best-case and worst-case.

Space Complexity

Space complexity deals with finding out how much (extra)space would be required by the algorithm with change in the input size. For e.g. it considers criteria of a data structure used in algorithm as **Array**or **linked list.**

How to calculate space complexity of an algorithm – https://www.quora.com/How-do-we-calculate-space-time-complexity-of-an-algorithm

Sorting



Sorting algorithms complexities'

Sorting is a skill that every software engineer and developer needs some knowledge of. Not only to pass coding interviews but as a general understanding of programming itself. The different sorting algorithms are a perfect showcase of how algorithm design can have such a strong effect on program complexity, speed, and efficiency.

Let's take a tour of the top 6 sorting algorithms and see how we can implement them in Python!

Bubble Sort

Bubble sort is the one usually taught in introductory CS classes since it clearly demonstrates how sort works while being simple and easy to understand. Bubble sort steps through the list and compares adjacent pairs of elements. The elements are swapped if they are in the wrong order. The pass through the unsorted portion of the list is repeated until the list is sorted. Because Bubble sort repeatedly passes through the unsorted part of the list, it has a worst case complexity of $O(n^2)$.

```
def bubble_sort(arr):
    def swap(I, j):
        arr[i], arr[j] = arr[j], arr[i]

    n = len(arr)
    swapped = True

    x = -1
    while swapped:
        swapped = False
        x = x + 1
        for i in range(1, n-x):
            if arr[i - 1] > arr[i]:
                swap(i - 1, i)
                swapped = True

    return arr
```

https://miro.medium.com/max/600/1*LllBj5cbV91URiuzAB-xzw.gif

Selection Sort

Selection sort is also quite simple but frequently outperforms bubble sort. If you are choosing between the two, it's best to just default right to selection sort. With Selection sort, we divide our input list / array into two parts: the sublist of items already sorted and the sublist of items remaining to be sorted that make up the rest of the list. We first find the smallest element in the *unsorted sublist* and place it at the end of the *sorted sublist*.Thus,we are continuously grabbing the smallest unsorted element and placing it in sorted ordering the*sorted sublist*.This process continues iteratively until the list is fully sorted.

```
def selection_sort(arr):
    for i in range(len(arr)):
        minimum = i

        for j in range(I + 1, len(arr)):
            # Select the smallest value
            if arr[j] < arr[minimum]:
                minimum = j

        # Place it at the front of the
        # sorted end of the array
        arr[minimum], arr[i] = arr[i], arr[minimum]

    return arr
```

https://miro.medium.com/max/1102/1*OA7a3OGWmGMRJQmwkGIwAw.gif

Insertion Sort

Insertion sort is both faster and well-arguably more simplistic than both bubble sort and selection sort. Funny enough, it's how many people sort their cards when playing a card game! On each loop iteration, insertion sort removes one element from the array. It then finds the location where that element belongs within another *sorted* array and inserts it there. It repeats this process until no input elements remain.

```
def insertion_sort(arr):

    for i in range(len(arr)):
        cursor = arr[i]
        pos = i

        while pos > 0 and arr[pos - 1] > cursor:
            # Swap the number down the list
            arr[pos] = arr[pos - 1]
            pos = pos - 1
        # Break and do the final swap
        arr[pos] = cursor

    return arr
```

https://miro.medium.com/max/1000/1*onU9OmVftR5WeoLWh14iZw.gif

Merge Sort

Merge sort is a perfectly elegant example of a Divide and Conquer algorithm. It simple uses the 2 main steps of such an algorithm:
(1) Continuously *divide*the unsorted list until you have *N* sublists, where each sublist has 1 element that is "unsorted" and *N*is the number of elements in the original array.
(2) Repeatedly merge i.e *conquer*the sublists together 2 at a time to produce new sorted sublists until all elements have been fully merged into a single sorted array.

```
def merge_sort(arr):
    # The last array split
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    # Perform merge_sort recursively on both halves
    left, right = merge_sort(arr[:mid]), merge_sort(arr[mid:])

    # Merge each side together
```

```
    return merge(left, right, arr.copy())


def merge(left, right, merged):

    left_cursor, right_cursor = 0, 0
    while left_cursor < len(left) and right_cursor < len(right):

        # Sort each one and place into the result
        if left[left_cursor] <= right[right_cursor]:
            merged[left_cursor+right_cursor]=left[left_cursor]
            left_cursor += 1
        else:
            merged[left_cursor + right_cursor] = right[right_cursor]
            right_cursor += 1

    for left_cursor in range(left_cursor, len(left)):
        merged[left_cursor + right_cursor] = left[left_cursor]

    for right_cursor in range(right_cursor, len(right)):
        merged[left_cursor + right_cursor] = right[right_cursor]

    return merged
```

https://miro.medium.com/max/600/1*fE7yGW2WPaltJWo6OnZ8LQ.gif

Quick Sort

Quick sort is also a divide and conquer algorithm like merge sort. Although it's a bit more complicated, in most standard implementations it performs significantly faster than merge sort and rarely reaches its worst case complexity of $O(n^2)$. It has 3 main steps:
(1) We first select an element which we will call the *pivot* from the array.
(2)Move all elements that are smaller than the pivot to the left of the pivot; move all elements that are larger than the pivot to the right of the pivot. This is called the partition operation.
(3) Recursively apply the above 2 steps separately to each of the sub-arrays of elements with smaller and bigger values than the last pivot.

```
def partition(array, begin, end):
    pivot_idx = begin
    for I in xrange(begin+1, end+1):
        if array[I] <= array[begin]:
            pivot_idx += 1
            array[I], array[pivot_idx] = array[pivot_idx], array[I]
    array[pivot_idx], array[begin] = array[begin], array[pivot_idx]
    return pivot_idx

def quick_sort_recursion(array, begin, end):
    if begin >= end:
        return
    pivot_idx = partition(array, begin, end)
    quick_sort_recursion(array, begin, pivot_idx-1)
    quick_sort_recursion(array, pivot_idx+1, end)

def quick_sort(array, begin=0, end=None):
    if end is None:
        end = len(array) - 1

    return quick_sort_recursion(array, begin, end)
```

https://miro.medium.com/max/600/1*hk2TL8m8Kn1TVvewAbAclQ.gif